A QUEL-to-SQL Data Manipulation Language Translator

by

John H. Webb

University of Canterbury
Christchurch
New Zealand

1988
Honours Project

Supervisor: Dr. N. Churcher

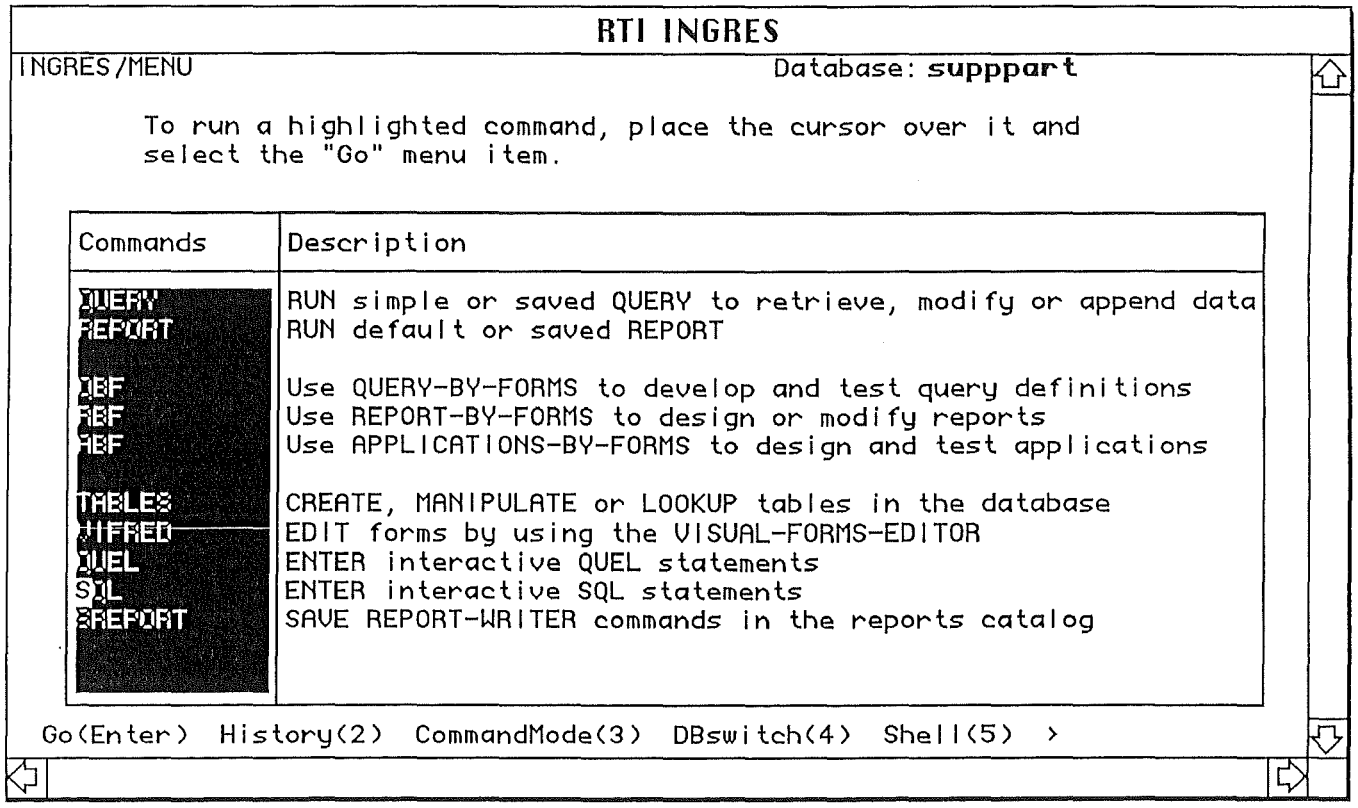


Table of Contents

Part I

Design Guide

Introduction

The field of database query language translation is not new. There are many applications where some form of query translation or query modification would be needed.

Katz and Wong[KATZ82] looked at a method for translating CODASYL data manipulation language (DML) to relational calculus. Their motivation was the problem of converting application programs that are applied to a new database system of a different level of procedurality.

Su and Reynolds[SURE76] developed an algorithm to convert queries when changes in the underlying database schema have occurred (e.g. splitting of large tables into many small ones, thus table access must be modified).

Owrang and Miller[OWRA87] have used the hypergraph representation of queries to develop an all-purpose method for translating between the relational and network data model. They did this as a means of providing communication between different database management systems (DBMS) without the user needing to know any of the foreign database's query language.

These studies have been carried out in various facets of query translation. Here we present an algorithm for translating between two different relational query languages - QUEL and SQL.

This document is not intended to be a tutorial in QUEL and SQL as such (though examples are given for each of the DML operations), instead the reader is referred to [DATE86]. However, we assume the reader has some working knowledge of the DML as in section 1 we detail some general translation principles that apply to the DML as a whole.

In each of the following sections 2, 3, 4 and 5 we take a detailed look at why we implemented the translator as we did and the steps we took in deciding how to translate the DML operators (retrieve, append, replace and delete respectively).

We also devote a section to the translation of aggregate functions (section 6) as this turned out to be a major stumbling block when translating to the 'equivalent' set functions. Translating aggregate functions was not fully implemented at the time this document was written as an error in the design was found. However the design decisions

for translating aggregate functions are important nonetheless and are documented here. Aggregate function translation proved to be a non-trivial problem.

In the second part of this report we document the implementation itself, looking at the Technical Guide and show how we were able to translate the queries. Accordingly we detail the implementation of each of the DML operations in a separate section (sections 7, 8, 9, and 10 deal with the retrieve, append, replace and delete respectively). We also show in a separate section the intended method for aggregate function translation.

In the final section we discuss the possible applications of this project and detail how it may be extended upon and enhanced in general.

The Relational Database

It was Date[DATE86] who said that database systems provide for centralized control of integrated and shared data. The advantages of this are that redundancy is reduced, inconsistencies are avoided and integrity maintenance is much more rigorous. In particular it is the relational database that has surfaced as being the most widely used due to its high level of data abstraction and sound theoretical base (the relational database is founded on the relational model[CODD70], unlike other non-relational systems - network, hierarchical, etc - which have no mathematical grounding).

The database query language is the only means by which the user may communicate with the database (either interactively or via the query language being embedded in some host language). In the relational system we have two flavours of query language. Those based on relational algebra[CODD72] and those based on relational calculus[CODD72]. Codd showed that relational algebra and relational calculus were not only functionally equivalent but also relationally complete[CODD72].

The Birth of QUEL and SQL

When the idea was first conceived of having databases based on the relational model, a number of major projects were undertaken in the early 1970's. One such project was the System R prototype developed by the IBM Research Laboratory, San Jose, California [ASTR76] (Codd himself worked here a few years before). Another was the INGRES prototype developed at the University of California at Berkeley. These prototypes were important in that they each spawned very different query languages.

System R was the first implementation of the language SEQUEL (later to become SQL) as defined by Chamberlin et al two years before[CHAM74]. SEQUEL was among the first implementations of the relational algebra. SQL has since been distributed by many vendors and has produced many dialects. SQL was designed to be easy-to-use for the novice and yet be relationally complete.

The INGRES prototype developed QUEL as it's query language. QUEL is opposite to SQL in that it is almost a pure implementation of the relational calculus. QUEL is not as idiosyncratic as SQL hence many relational databases support a QUEL-like query language. There is now a commercial version of INGRES produced by Relational Technology Incorporated (RTI INGRES) that supports SQL and QUEL as a query language.

Aims and Objectives

As was stated earlier, there are many reasons why we might need a translator of query languages. With SQL about to become the industry standard (as announced by ISO - (ISO TC97 / SC21 N1479)) and QUEL being not only widespread but also having historical significance, we present an algorithm for translating QUEL to SQL.

Were the translation process simply one of syntax this task would be trivial. However it is that QUEL and SQL are of different backgrounds - QUEL based on relational calculus and SQL based on relational algebra -

that provides the challenge. It is the semantics of the query we wish to translate.

Here we consider the data manipulation language (DML) in particular, focussing our attention on whether there is some set of queries expressed in QUEL that are not are not able to be expressed in SQL. If QUEL was a pure implementation of the relational calculus and SQL was a pure implementation of the relational algebra we would expect that every QUEL query would have some equivalent SQL query by Codd's theory of functional equivalence. However as SQL is more a hybrid of relational algebra and relational calculus this may not be the case. Thus our objective is to see how well an implementation of the relational algebra SQL is.

The General Overview

To show the differences in each language and to gain a practical insight into the problems of translating queries from University INGRES QUEL to RTI INGRES SQL a translator was constructed with the aid of the compiler writing tools Yacc and Lex[SCHR85]. The resulting translator takes a stream of ASCII characters as input and outputs the resulting SQL equivalent (as a stream of ASCII characters) to the standard output. (Note: this translator may be applied to the standard SQL with very little modification as there is little difference between the ISO version and the RTI version of SQL[FERR87] as far as the DML is concerned).

The translator was developed with the intention of interfacing with the INGRES database. With this in mind the translator does not deal with any complex tasks like query optimization (except for removing aliases) as the target SQL system would do all of these functions and more.

By using the tools described above to create the parser more effort could be spent concentrating on the translation itself. To aid in this, every query that was given as input to the translator was decomposed and restructured into a parse tree. The general strategy of the translation process was to try and take advantage of the mixed nature of SQL and

convert the original QUEL query into an equivalent calculus-based SQL query.

Should the translator detect an error in the input then it immediately reported it, printing the illegal token (and the line on which it was found) to the standard error. When such an error occurs then the translation is immediately aborted and the contents of the original QUEL query lost.

In building the translator it was hoped to meet the proposed aim of identifying some set of queries in QUEL that have no equivalent in SQL. If we were able to show that there were some queries that may not be translated then we may draw a number of conclusions:

1. SQL is not a pure implementation of the relational algebra
2. QUEL is not a pure implementation of the relational calculus
3. Despite their relational backgrounds QUEL and SQL are just too different, too idiosyncratic in some cases such that there is no equivalent expression of a query.
4. Codd's paper detailing the functional equivalence of relational algebra and relational calculus are essentially wrong

The translation of the data manipulation language itself was most successful, despite the fact that the translation of the aggregate functions were not completely satisfactory. All general queries may be handled in this translator for almost all cases of data manipulation.

The findings that resulted from the translation were significant in that a class of QUEL queries were discovered to have no equivalent translation. Of the four possible explanations that were outlined it is felt that the reason for this may be attributed to the fact that SQL is not a pure implementation of the relational algebra but merely some part of it.

Section 1

Translation of the Data Manipulation Language

A database query language is comprised of two sublanguages - the data definition language (DDL) and the data manipulation language (DML). The data definition language is the sublanguage that is used to describe the format of a table that will store data. The data manipulation language is the sublanguage that allows the user to manipulate the data stored in said table. Here we look at the problems involved in the translation of the DML.

There are four types of DML operators:

Data Retrieval - for retrieving data from tables
Data Deletion - for deleting data from tables
Data Modification - for updating values stored in the tables
Data Insertion - for inserting data into the tables

Both QUEL and SQL have operators that perform these tasks. But the point to note is that they each have their own method of implementing these operations. That is, each of the corresponding operators differ quite dramatically in the underlying methods of implementation.

General Translation Strategy

In considering the translation from QUEL to SQL we take note of some of aspects that occur to the general translation process as a whole. It is worth mentioning that the source-to-source translation of query languages is different in many respects to the translation of lower-level,

more procedural languages. For instance, the kinds of things that may be translated are fewer. Things such as type, memory size and addresses, etc. do not need to be collated, translated, and passed to the target system. Theses are irrelevant to the translation of the high-level DML. Yet there are also similarities to translating between low-level languages in that we must retain parsing information. To retain this information we store it in a parse tree.

Treatment of Aliases

One point that should be made about translating any DML query is that the range variables (table aliases that may be specified in the QUEL query) are not translated to their equivalent in SQL. They are instead replaced by the actual table names they represent. This is done purely for efficiency reasons (the target SQL system does not have to do the replacement itself). Obviously this will not alter the semantics of the query as range variables merely provide a convenience for the user.

e.g.
```
range of s is supplier
range of p is parts
retrieve (s.snum, p.pnum)
      .
      .
```

is translated to

```
select supplier.snum, parts.pnum
from supplier, parts
      .
      .
```

Another thing to note is that should a range variable reappear in another range statement then the latter declaration is ignored.

e.g.
```
range of s is supplier
range of s is shipment
```

Any reference to s in the QUEL query is taken to be referring to the the supplier statement.

Explicit Qualification Retained

In QUEL all references to the columns of a table must be qualified with the name of the table it comes from. This explicit qualification is retained in the translated query for all column names. This is legal though, for the most part, unnecessary as implicit qualification of columns is a feature of SQL. However there are two aspects that dictate the need for explicit qualification in SQL:

1. If a query target list involves some targets that are table wide access and some targets that are column access.

   e.g. `retrieve (a.all, b.x)`

   .
   .

   we would want to qualify the table which we want to access all columns for:

   ```
   select a.*, x
   from a, b
   ```

   .
   .

2. If two or more tables are listed as the target of a query, we have no way of deriving from the QUEL query alone whether or not a column is common to many tables:

   e.g. `retrieve (a.x, b.y)`

   .
   .

   may be readily translated to

```
select x, y
from a, b
```

but if 'a' and 'b' have a column 'x' or 'y' (or both) common to both tables then an ambiguous column reference will result. We have no way of checking this.

Thus it is this last point in particular that decides that explicit qualification is necessary for all column references.

Conclusion

As we have stated before SQL is somewhat a mixture of relational algebra and relational calculus. Thus we endeavour as part of the overall translation strategy to take advantage of this by translating the QUEL query to a calculus equivalent of SQL where possible. We now may consider the translation of each of the DML operators.

Section 2

Data Retrieval

Within a single data retrieval statement we have all the functionality of most of the algebraic operators first devised by Codd[CODD72]. Also included are the facilities for sorting and duplicate value removing (and grouping in the case of SQL). Let us take each of the optional clauses of the RETRIEVE statement and show the resulting translated query.

```
RETRIEVE [ [INTO] <table_name> ]
        [UNIQUE] ( <target_list> )
        [WHERE <predicate> ]
        [SORT [BY] <sort_item_list> ]
```

Fig. 1 The retrieve statement of QUEL

```
subselect { UNION subselect }
[ ORDER BY column [ASC | DESC] {, column [ASC | DESC] } ]
```

where a subselect has the syntax:

```
SELECT [ALL | DISTINCT] expression {, expression }
 FROM tablename [corr_name] {, tablename [corr_name] }
[ WHERE search_condition]
[ GROUP BY column {, column} ]
[ HAVING search_condition ]
```

Fig. 2 The select statement of SQL

The simplest retrieval we may have is that of some table(s) without any restriction. The translation for this is simply a matter of syntax:

```
e.g. retrieve (a.x, b.y, c.all)
```

translates to

```
select a.x, b.y, c.*
from a, b, c
```

Similarly, the translation for retrieving unique values is also a simple matter:

```
e.g. retrieve unique (a.x, b.y)
                .
                .
```

translates to

```
select distinct a.x, b.y
from a, b
       .
       .
```

The Target List

The target list of the RETRIEVE statement may contain references to any table in the database and any of the aggregate functions. These may be contained in any number of scalar expressions. The target list may be passed on to the select clause with little or no modification (however, dealing with aggregate functions is a non-trivial matter. For treatment of aggregate functions see section 6).

```
e.g. retrieve (a.x, b.y, total = a.x + b.y)
                .
                .
```

may be translated to

```
select a.x, b.y, total = a.x + b.y
from a, b
        .
        .
```

However to translate the following is not trivial:

```
retrieve (a.x = min(a.y by b.y where b.y < 42) )
```

The WHERE clause

The translation of the predicate contained in the WHERE clause is one place where we take advantage of the calculus nature of SQL. That join predicates need not be modified to fit some algebraic form means that the WHERE clause may be appended to the SELECT clause virtually unaltered (again see section 6 on aggregate functions for cases where the where clause must be modified).

e.g.
```
retrieve (a.x, b.y, c.all)
    where a.x = b.x
    and b.z = c.z
```

is merely translated to

```
select a.x, b.y, c.*
from a, b, c
where a.x = b.x
and b.z = c.z
```

Sorting the Data

There is one main difference that must be noted between QUEL and SQL when it comes to sorting the results of a retrieval. In SQL, data may be sorted by either an unqualified column name or by column number. QUEL on the other hand may sort only by qualified column reference. In translating the SORT BY clause into the equivalent ORDER BY clause we

currently use the column name. Although it would not take too much effort to change the reference to a column number we have yet to do so. Obviously it would be best to order by column number as only then may we retain the exact meaning of the original query. To illustrate the differences between ORDERing BY a column number and ORDERing BY a column name we consider the following:

```
e.g. retrieve (a.x, b.y, c.x)
                .
                .
     sort by b.y, c.x
```

This may be translated into a form with an unqualified column name thus:

```
select a.x, b.y, c.x
from a, b, c
        .
        .
order by y, x
```

However this is not what the original query specified. This resulting query implies we sort by `b.x, c.x` and `a.x`! Thus we have lost some of the intent of the original query. Hence all columns of the ORDER BY clause must be translated to an equivalent column number as specified in the SELECT clause:

```
select a.x, b.y, c.x
from a, b, c
        .
        .
order by 2, 3
```

Storing the Results

Finally we address the question of storing the results of the query. When retrieving data as we have in previous examples the results of the query would output to the terminal monitor (unless being accessed though EQUEL, the embedded query language of QUEL). If we wish to store the results into some other table of the same dimensions we may do so in QUEL by giving a table name argument before the target list. This will create a table with this name and then, after executing the data retrieval, put the results INTO that table. In the INGRES version of SQL we approach the storage issue in the same way but we must do so in an algebraic form.That is we create the table as a result of a subquery.

```
e.g. retrieve into AA (a.x, b.x, b.y, c.z)
     where
             .
             .
```

is translated to

```
create table AA as
  select a.x, b.x, b.y, c.z
  from a, b, c
  where
        .
        .
```

That is, we take the original query and translate it into a subquery of a CREATE statement. Only in this way may we create a new table in SQL in a manner that matches the QUEL INTO clause.

Redundant Features in the Translation

Here are the SQL features that are redundant in translating the RETRIEVE statement as they have no equivalent statement in the QUEL syntax:

- HAVING clause

The having clause is used primarily for putting references to set functions as they pertain to the main query. However this is not needed during the translation of a QUEL retrieve statement as all HAVING -type functions are handled by the WHERE clause in QUEL. Why, then, have we not used the HAVING clause for translating aggregate functions to set functions? Because the aggregate functions are much more powerful than their SQL set counterparts. The HAVING clause does not allow the same power as the aggregate functions.

- The GROUP BY clause

The GROUP BY clause takes the result of the query and groups them together by some column. This is also not needed as there is no semantic equivalent. (However, they can be used in translating aggregate functions in QUEL to set functions in SQL).

- The ASC | DESC options

In QUEL there is no way to specify that the results of the retrieval will be sorted in the direction of the users own choosing. Thus such an option is not necessary in translating from QUEL to SQL.

- The UNION clause

This is strictly a feature of SQL and is the only operator that is an explicit relational algebra operator (the SELECT clause can also be an explicit relational algebra operator but is much more besides) . Thus as there is no algebraic operations in the calculus-based QUEL, this clause will never be needed for translating a query.

Section 3

Data Insertion

Here we look at the problems involved in translating from the APPEND statement of QUEL to the INSERT statement of SQL.

APPEND [TO] tablename (target_list) [WHERE qual]

Fig 1. The syntax for the QUEL append

INSERT INTO tablename [(column {, column})]
[VALUES (expression {, expression }) | subquery]

Fig 2. The syntax of the SQL insert

The Design Problem

The best way one may translate the APPEND clause to the equivalent INSERT is to use the algebraic subquery. This was decided upon because the VALUES clause did not permit the use of any arbitrary expression. An expression that may contain references to aggregate (or rather set) functions, or other tables. Thus this automatically restricts the kind of VALUES one is able to store (as set out in the APPEND target list) to only the trivial cases.

```
e.g. append to a (x = 42, y = 42)
```

may be readily translated to

```
insert into a (x, y)
values (42, 42)
```

To refer to more complex examples that make reference to some other tables we may only do so by using the algebraic subselect statement.

e.g. `append to a (x = 42, y = b.y)`

we must use the subquery option

```
insert into a (x, y)
  select a.x, b.y
  from a, b
```

Fortunately, we need not use the column list when we make use of the subquery option. Thus we may generalize all queries to a form where we may use the subquery thus:

e.g. `append to a (x = 42, y = c.y + b.y)`

may be translated to

```
insert into a
  select x = 42, y = c.y + b.y
  from b, c
```

Note also how we are able to implement the WHERE clause from the APPEND query by attaching it to the new subquery.

```
e.g. append to a (x = 42, y = c.y + b.y)
     where b.y = 42
     and
             .
             .
```

may be translated to an equivalent subquery:

```
insert into a
  select x = 42, y = c.y + b.y
  from b, c
  where b.y = 42
  and
        .
        .
```

Thus we may make use of the subquery option for translating data insertion statements by converting all insertion queries in QUEL to an algebraic form. Note, however, how we are again able to take advantage of the calculus side of SQL's subselect statement. Because of this we are able to transplant the WHERE clause of the APPEND statement onto the subquery without any need for further modification (see section 6 for a discussion on aggregate functions. In this case further modification is needed).

Notice how we do not have the problem of a column name being common to many tables (and hence ambiguity of unqualified columns) as the target list takes on a different meaning in this new context. Whereas before all column references in the target list were all implicitly qualified by the Table a (that is, they were column references of table a), within the subquery's SELECT clause they are the new names of columns to be retrieved. Also there is no problem of ambiguous column names in the scalar expressions as QUEL ensures that column references here must be qualified.

That we are not forced to specify the column list is significant in that we may apply this schema completely, no matter what the query target list involves. The advantage is evident when we consider the following:

```
e.g. append to a (b.all)
        .
        .
```

which we may easily translate to

```
insert into a
  select b.*
  from b
       .
       .
```

Were we forced to list all columns the translation for this particular kind of query would not be possible as we would have no way of knowing what columns table `b` had.

Redundant Features in the Translation

With every APPEND statement being able to be translated to an algebraic INSERT INTO statement we need never use the VALUES clause. This is because we are able to specify all insertion queries in an algebraic manner, even those insertions of a simple form. However the reverse case, of being able to use the simple form for any arbitrary insertion, will not necessarily hold.

Section 4

Data Modification

Here we look at the problem of translating the data modification operations from QUEL to SQL

```
REPLACE row_variable (target_list) [WHERE qual]
```

Fig 1. The syntax of the REPLACE statement in QUEL

```
UPDATE tablename [ corr_name ]
SET column = expression {, column = expression }
[ WHERE search_condition ];
```

Fig 2. The syntax of an UPDATE statement in SQL

The translation of the replace for the simplest of cases is fairly trivial:

```
e.g. replace a (x =  42)
```

is simply translated to become

```
update a
set x = 42
```

For the more complex examples we are able to use the WHERE clause, translating thus:

```
e.g. replace a (x = 42)
     where a.y < 42
```

may be translated to the equivalent

```
update a
set x = 42
where a.y < 42
```

and the more important translations may be done thus:

```
replace a (x = 42)
where a.y < b.y
```

is translated to the include the subquery

```
update a
set x = 42
where a.y < any
  (select b.y
  from b
  where a.y  <  b.y)
```

Again we are able to accommodate this fairly easily with the aid of the subselect statement that must be used whenever reference to another table is made. Let us look at the result of what happens when another table is included in the target list.

e.g. `replace a (x = b.x)`

In this instance we are unable to translate the query to an equivalent in SQL as the SET clause does not permit the reference of other tables! This is somewhat disappointing but there is no way to readily translate queries of this form without knowing more about the very nature of the tables themselves.

The Problem

The problem here is due to the SET's inflexibility in not allowing references to other tables. This means that we must emulate the situation

when such a reference occurs in the REPLACE target list. The only way we may emulate the update condition is through some variation of deleting the rows of the table that meet some condition, then inserting a new set of rows with the updated values. To show that this will in fact not work we consider the following scenario:

```
e.g. replace A (x = B.x)
     where A.y = B.y
```

Append a new row on to the end of the table with the necessary conditions from the REPLACE target list:

```
insert into A
  select x = B.x
  from A,B
  where A.y = B.y
```

Then carry out the update referencing this new row:

```
update A
set x = A.x
where A.x in
  (select A.x
  from A
  where A.y = B.y)
```

Then delete the row from the table once update has been completed

```
delete from A
where A.x in
  (select A.x
  from A
  where A.x = B.x)
```

In principle, at least, this would work but it is not a general enough algorithm to handle all cases. Consider the following tables.

Table a:

x	y
1	s
2	s
3	r

Table b:

x	y
4	s

Following the steps of the algorithm we first insert a new row into Table a with the value of b.x:

Table a:

x	y
1	s
2	s
3	r
4	

We then update the values of the a.x:

Table a:

x	y
4	s
4	s
3	r
4	

We then delete the table of the temporary row we inserted. But in doing so we delete the table of more than the last row. Indeed we delete the very rows we have just updated!

Alternatively we might try the following:

1. Create a new table of the same dimensions of Table A but with one extra column from Table B.

   Table Temp:

x	y	z

2. Copy in the details of the old Table A and a new blank column for the result for column B

   Table Temp:

x	y	z
1	s	
2	s	
3	r	

3. Append a new row that contains the row of Table B that meet the constraint for A.y = B.y this time putting a value in for z.

   Table Temp:

x	y	z
1	s	
2	s	
3	r	
4	s	4

4. Make the UPDATE from this newly appended row.

Table Temp:

x	y	z
4	s	
4	s	
3	r	
4	s	4

5. Delete the temporary row.

Table Temp:

x	y	z
4	s	
4	s	
3	r	

6. Delete the temporary column.

Table Temp:

x	y
4	s
4	s
3	r

However good this idea would be in theory, in practice this is not possible. Firstly, it is not possible to create the table with a new column without inserting values into it. What values should these be? We might decide to put the values of B.y in but when we come to make the UPDATE we do not have a unique row reference which we might use. We may instead think about putting a zero value in for the column z and make a non-zero reference value for z when we append our temporary row. But here we are assuming that the field is numeric.

Thus one may extend this and see that no matter how one chooses to insert, update and delete rows of a table (or variations thereof) one cannot correctly emulate the REPLACE statement when another table appears in

the target list. This also means that we are restricted in emulating the REPLACE statement where we have an aggregate function appear in the target list. This is in spite of the way we handle aggregate functions.

To conclude, then, we may not translate queries of this nature. This is not to say that such a REPLACE statement is impossible to emulate. Merely we conclude that a query of this nature can not be translated given the information derivable from the query. Should such a query be passed to this translator the update of such a column will be ignored. This restricts the QUEL user somewhat in that they must reformulate the original query.

Section 5

Data Deletion

Here we look at the problem of translating the data deletion statements from QUEL to SQL.

DELETE FROM tablename [corr_name]
[WHERE search_condition]

Fig 1. Delete Statement in SQL

DELETE row_variable [WHERE qual]

Fig 2. Delete Statement in QUEL

The case for translating the query where the user is deleting a whole table is trivial:

e.g.
```
delete A
```

is merely translated to

```
delete from A
```

Likewise, if we are deleting a table with a local restriction (that is the restriction involves only some range of itself) then we may attach the constraint on to the resulting SQL query:

e.g.
```
delete A
where A.x = 42
```

is translated trivially to

```
delete from A
where A.x = 42
```

However, the search_condition in the WHERE clause may not contain reference to another table unless it is contained in a subquery. Thus any WHERE clause of the QUEL query that contains reference to another table must be modified to include an (algebraic) subquery.

e.g.
```
delete A
where A.x < B.x
and
        .
        .
```

is translated to

```
delete from A
where A.x < any
  (select B.x
  from B
        .
        .
```

The any query may be used to good effect by applying it after the comparison operator in the QUEL predicate.

The Use of the Subquery

The basic premise in translating the delete statement, then, is simply a matter of checking whether or not a QUEL query contains any reference to another table. If there is no other table we pass the original WHERE clause unaltered. Otherwise we look for a join predicate involving the main table (the table being deleted from).

With this in mind it would have been perfectly legal to translate every join predicate into a subquery, thereby having a as many nested subqueries as there are join predicates.

e.g.
```
delete A
where A.x = B.x
and B.y =
   .
   .
   .
and C.y = 42
```

might be translated to become

```
delete from A
where A.x = any
  (select  B.x
  from B
  where B.y = any
        .
        .
        .
        where C.z = 42) ... ) )
```

Initially, this method was used but due to its complexity it was abandoned. The scheme involved the following:

1. Simply take a copy of every linkage condition.
2. Sort the condition into the correct order such that each one may put the links into an appropriate order.

```
retrieve (A.x)
where D.x = B.y
and C.z = D.z
and  A.p = D.p
```

leftlink	rightlink
D.x	B.y
C.z	D.z
A.p	D.p

```
select A.x
from A
where A.p = any
  (select D.p
   from D
   where D.z = any
     (select C.z
      from C
      where D.x = B.y)
```

leftlink	rightlink
A.p	D.p
D.z	C.z
D.x	B.y

This is not only unnecessary but also extremely difficult. The major problem in attempting to translate in this fashion is that one must find all the necessary join predicates and then sort them into the correct order. (By a necessary join predicate we mean one may be linked back to the main table either directly or indirectly). To do this is extremely difficult in practice.

Section 6

Aggregate Functions

In dealing with aggregate functions we considered the differences between them and their SQL set counterparts. The main differences are outlined here:

- QUEL aggregate functions may be nested within each other while SQL set functions can not.
- a QUEL aggregate function may occur within either the WHERE clause or the target list. Compare this with set functions in SQL which may only occur in the SELECT clause or the HAVING clause.
- aggregate functions have a greater flexibility than set functions, being able to specify grouping and other predicates that are completely independent of the main query.

Thus to translate the full power of the QUEL aggregate functions we must find some structure that will emulate all of these conditions.

Although we were not successful in completing the translation of aggregate functions at the time this report was written, we detail the steps that were undertaken to reach our yet-to-be-fully-implemented final design.

The Initial Design

Our first design choice was to use a system of subqueries as a means of representing the aggregate function. This was chosen because the subselect allowed us to have all the groupings and independent conditions that could be found in the QUEL aggregate functions.

e.g.

```
.
.
max( A.x by A.y, B.x where B.x < 42
                    and       .
                              .     )
```

could be easily translated to become

```
.
.
(select max(A.x), A.y, B.x
 from A, B
 where B.x < 42
 and         .
        .
 group by A.y, B.x )
```

We would also be able to emulate nested aggregate functions by simply nesting the subqueries. Thus we have the functionality of an aggregate function allowing us the freedom to nest the set functions and allow them to appear in the WHERE clause of the query. However the problem still remains for dealing with an aggregate function occurring in the target list.

To emulate this we would require the GROUP BY applied to the main query. Thus we might have:

```
retrieve (A.a = min( A.b by B.b) ... )
```

which might be translated to

```
select A.a = min (A.b), B.b
from A, B
.
.
group by A.a, B.b
```

Unfortunately (as was realised after implementation had begun) this means that the grouping applies to the entire query, not just the aggregate function alone. Also the WHERE clause that would be appended would be an added constraint to the table as a whole. This is not acceptable.

Thus the subquery is not a good tool to choose for emulating the aggregate functions as they are too cumbersome and not general enough to cater for all cases. But we still want the same flexibility and power of the SELECT clause. Thus we choose instead to create a new temporary table for every occurrence of an aggregate function in the original query and replace said occurrence with a reference to this new table.

The Final Design

The beauty of this solution is that we convert the problem of aggregate function handling to a more familiar one of dealing with tables. By close inspection can see that this scheme will allow us to replace the occurrence of an aggregate function no matter where the function occurs.

But it is the handling of nested aggregate functions that deserves special attention. Nesting of aggregate functions may be emulated thus:

e.g. `... avg( sum(A.y by B.y) ) ...`

may be translated to the following (these would be output before the main query).

```
create table agg0 as
  select result = sum(A.y), B.y
  from A, B
  group by B.y;

create table agg1 as
  select result = avg(agg0.result), agg0.y
  from agg0;
```

Note the order of the declarations - the innermost nested function listed first. Now the reference to the nested function will be replaced by a simple

reference to

```
agg1.result
```

Finally, one would need to generate explicit linkage conditions from these new tables to the main query (In the example above, the reference to B.y is an implicit link between the aggregate function and the main query. Hence the main query, along with having reference to this new table will have the where clause modified to include an explicit linkage condition.

e.g.
```
retrieve (A.x, B.x)
where A.x < min(B.x by B.y)
```

will be translated to become:

```
create table agg0 as
     select result = min(B.x), B.y
     from B
     group by B.y;

select A.x, B.x
from A, B
where A.x < agg0.result
and B.y = agg0.y;
```

Finally, once the query was translated in full we would need to remove the temporary tables created. To do this we issue a DROP command for all of the tables we made.

```
drop agg0;
drop agg1;
```

Thus, although this scheme is not fully in place, we have outlined how we would be able to translate aggregate functions. The algorithm for doing so is simply a matter of converting the problem to one of dealing with aggregate functions to one of dealing with table - a problem we are already equipped to deal with.

Part II

Implementation Guide

Technical Notes

Introduction

Now that we have seen what design decisions were made we can now look at how the translations were implemented. In this part we will look at each of the statements in turn and show the mechanics involved in translating a QUEL query into an SQL query.

The Translation Tools

Th translator was written using the compiler writing tools Yacc and Lex[SCHR]. Lex is a lexical analyser or scanner. By describing the form of each legal input token that may be contained in a QUEL query it scans the an input stream checking for illegal input symbols. In short it is a 'word recognizer'. Yacc takes the input seen by Lex and decides whether the input meets the required syntax of a QUEL query. In short it is a 'language recognizer'. Together they check that a QUEL query is valid.

Parsing

As previously stated Yacc is a language recognizer. Hence the least we could expect from using Yacc is to ensure that a QUEL query that is passed into our translator is syntactically correct.

However it is up to the programmer to provide data structures, routines, etc for handling the translation mechanism itself. The most

appropriate form of storing the input, ready to make changes to, etc is in a tree structure. A tree such as this that is used for translation purposes is known as a parse tree.

Within this translator there are often many such trees that are being constructed for a particular aspect of the original query e.g. the tree for the WHERE clause, one for the target list, etc.

The translation process, then, is the manipulation and modification of these trees.

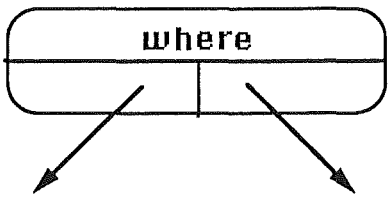
Parse Tree Construction

To translate from one query language to another one must store the source query in a form that may readily be manipulated. For this purpose a parse tree is used. A parse tree is a convenient data structure that allows the dynamic storage of input. As more input is received, the larger the tree grows. Once the input is exhausted we may set about modifying the tree to the target language.

In this translator we put every token read into a tree node with allowed space for two children. Thus the input

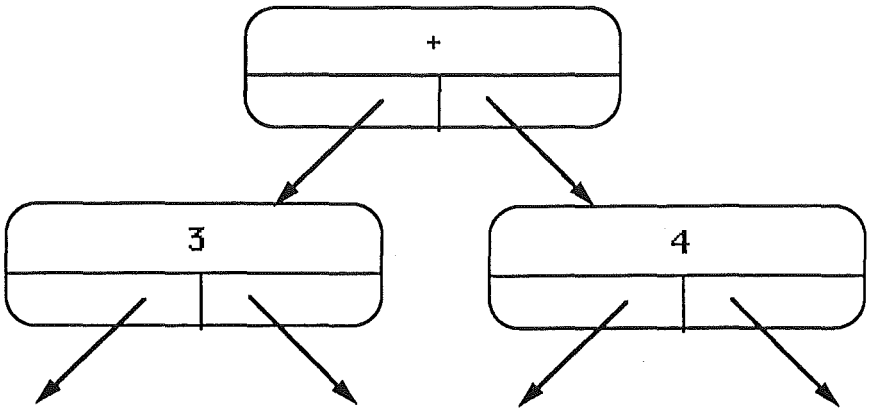
```
" ... where ..."
```

might be stored in a tree node like this



Also it should be noted that the tree is constructed and traversed in an inorder fashion. By this we mean that the tree is constructed such that it

may be scanned by inorder (visit the left child, visit the root then visit the right child of the tree) traversal. Thus part of a scalar expression containing the input "3 + 4" might be stored as



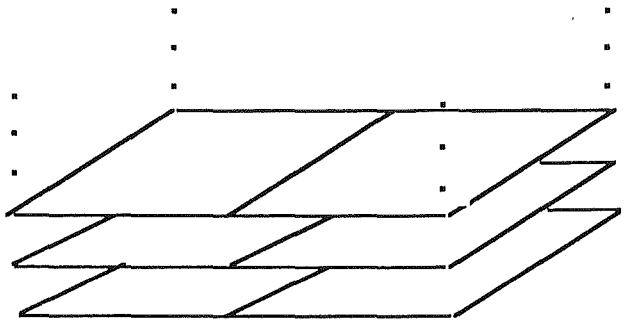
Now that we know what main data structures are used and how we make use of them we may now address the question of query construction and show how the translations are performed.

In translating the query the general pattern adhered to was to translate as much as possible as soon as possible. That is, if while parsing the original QUEL query we had enough information to make the translation (in mid-stream as it were) we would do so. Instances where this would happen include the translation of the syntactic differences (e.g. the translation from "all" to "*") and some of the major translations (e.g. if an aggregate function was discovered then it was translated immediately before reading the rest of the query).

Aliases

Aliases, or range variables, were not kept in parse tree form, rather they were kept in a separate stack. Every new occurrence of a range variable declaration or a reference made to some table not mentioned in a range statement was added to the alias stack. All recurring range variable declarations were ignored. The point to note with aliases is that

they may occur any where. Thus it was for this reason that a stack, it was decided, would be more natural and much easier to maintain.



The ALIAS stack
one cell for the range variable and one for the table name

One aspect worth pointing out is that all FROM lists generated in the resulting SQL query have the table names listed in reverse order to that which they were given in the list of range variable declarations in the QUEL query. This is one of the tell-tale signs that the aliases are stack-based in that the aliases are popped from the stack in reverse order. Since this does not alter the semantics of the resulting query in any way this was not considered necessary to rectify.

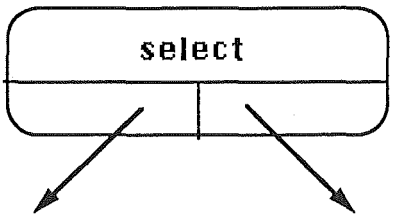
Section 7

Data Retrieval

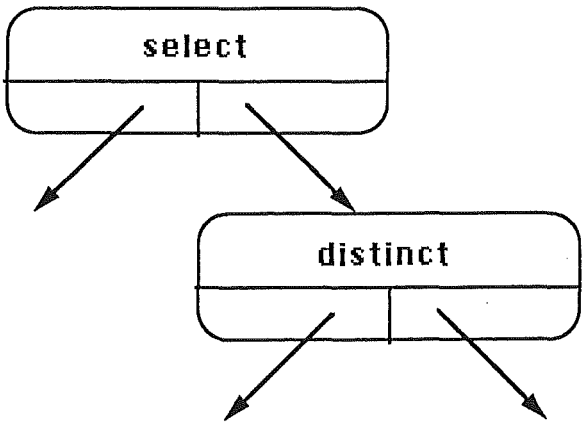
Here we shall look at the different options that are in the RETRIEVE statement and detail how each was translated.

Translation does not begin until after the RETRIEVE statement has been parsed fully and the various parse trees constructed. Once this is done the following steps are undertaken:

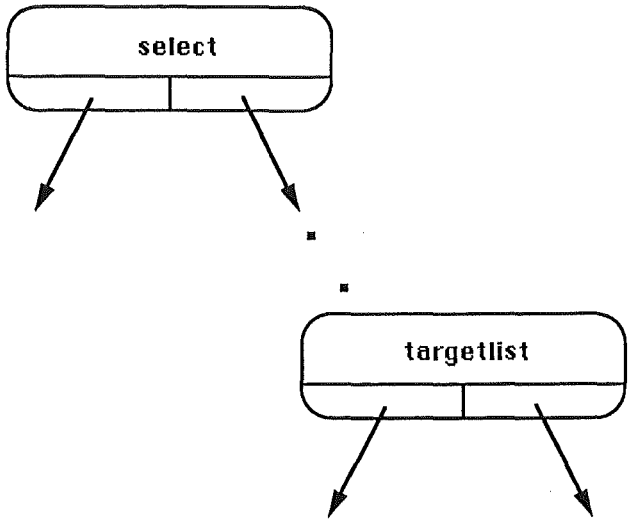
1. Firstly the query tree was constructed, containing the simple SELECT statement.



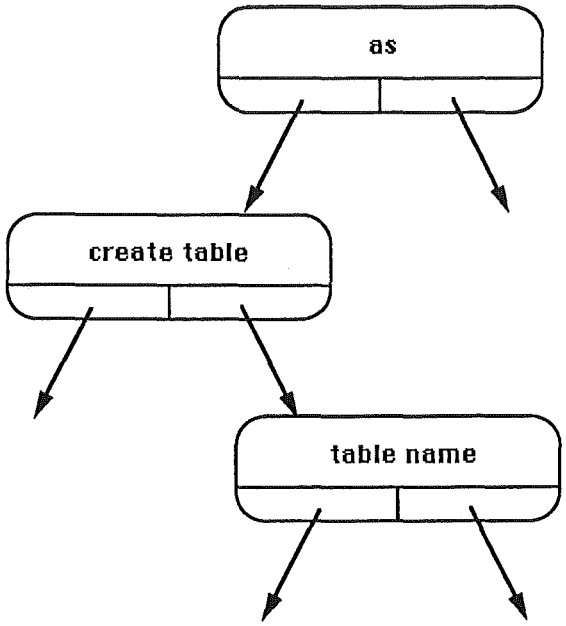
2. The unique flag is checked to see if a DISTINCT word should be inserted into the tree. If so, it is placed to the right of the SELECT clause.



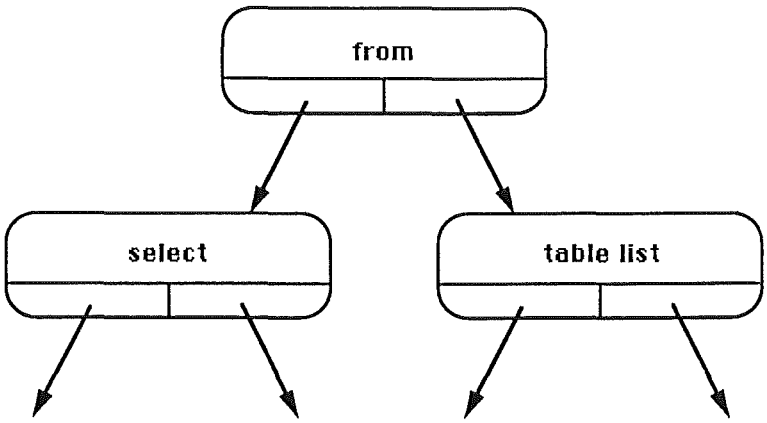
3. If there is a target list it is inserted to the right of the SELECT clause. (After the DISTINCT if there is one).



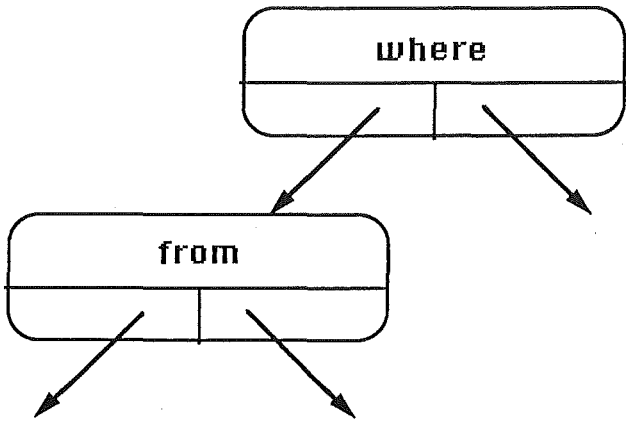
4. If a table name is given (that is, we are retrieving into a table) then we construct a subtree that contains the necessary CREATE clause. This subtree is attached as the left child of the SELECT node.



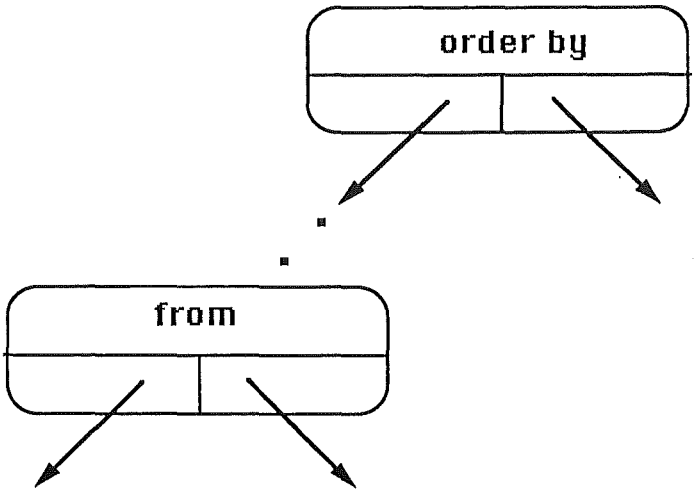
5. a FROM clause is generated, with contents of the alias stack being condensed into a single string and placed to the right. The SELECT clause is now made the left child of the FROM clause.



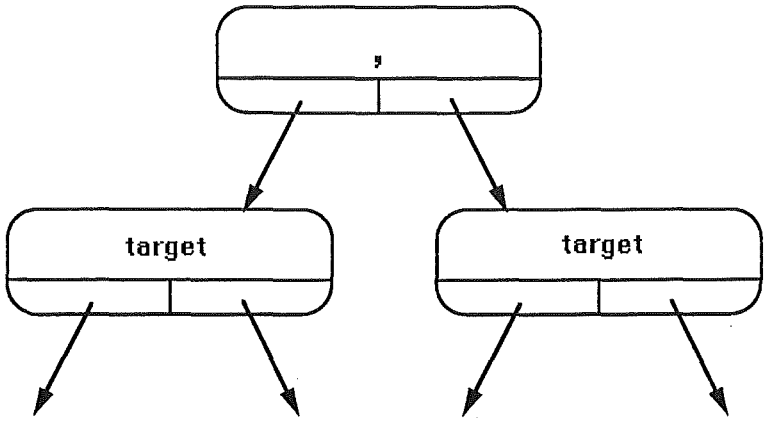
6. If there is a WHERE clause we make this the parent of the FROM clause, connecting it to the left child of the WHERE subtree.



7. If there is a SORT BY clause then we make this the root of the tree with the WHERE subtree (or FROM subtree if there is no WHERE clause) made the left child.



Thus with this strategy we may construct any SELECT statement from the RETRIEVE. The target list is a comma separated list of scalar expressions and/or column references.



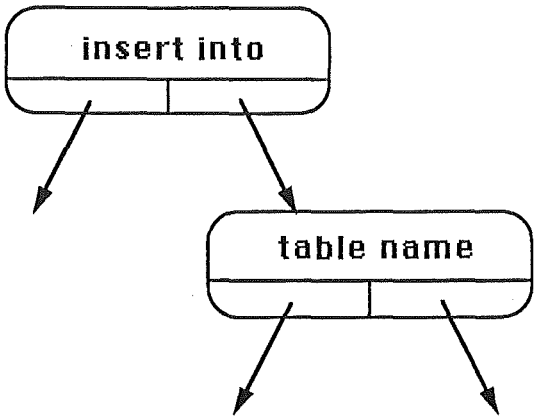
Section 8

Data Insertion

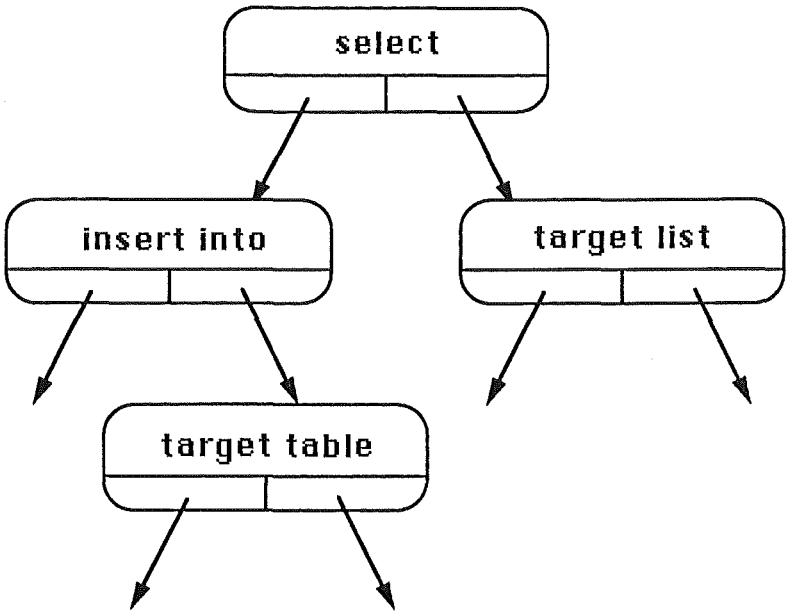
With the APPEND statement we have taken the option to translate it to an algebraic form for all data insertion queries. For this instance, however, we are able to do so with greater ease than was the case for other DML translations.

1. If the target name is an alias we replace it with the actual table name.

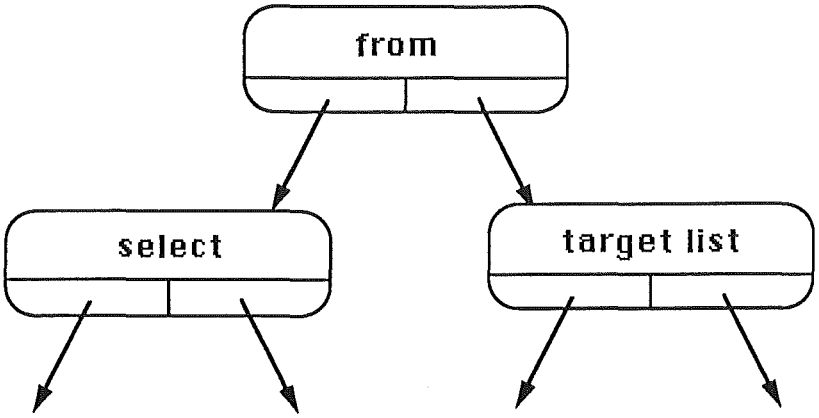
2. Now we build the target relation. First we connect the table name to the INSERT INTO clause.



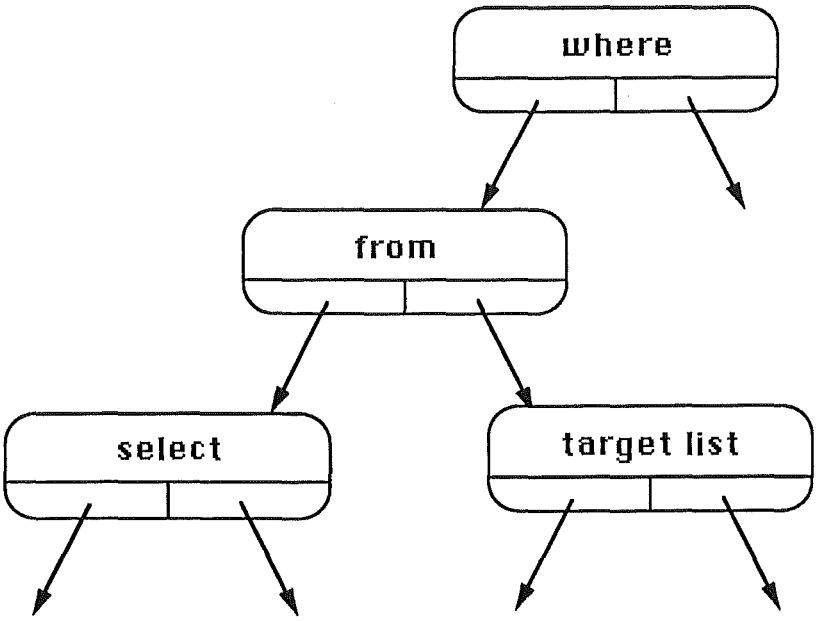
3. Next we connect the subquery template itself. This is different to the other subquery templates in that we do not specify access through a join predicate. Indeed we do not need to search the WHERE clause for a join predicate at all. This simplifies the translation process a great deal.



4. To complete the template we then include the FROM clause.



5. Finally we check that for whether there is a WHERE clause to apply. If so, we append this constraint to the subquery.



Thus we have shown how the original QUEL APPEND statement may be stored in a (parse) tree structure and then once stored we may manipulate this tree, reshaping it to become the resulting SQL query.

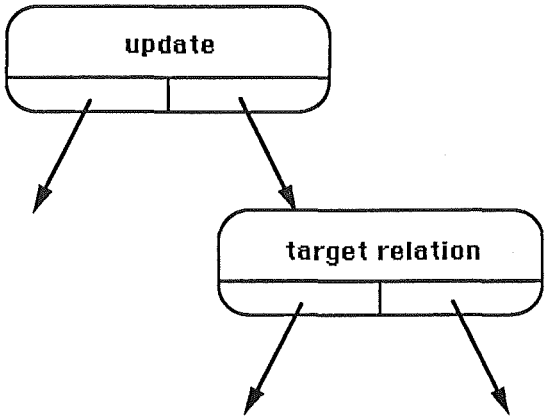
Section 9

Data Modification

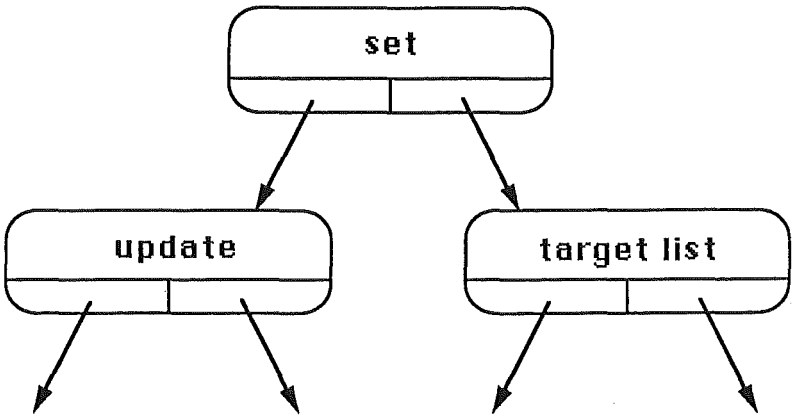
Here we detail the mechanics involved in translating the REPLACE to the UPDATE statement. As we saw in the design section there are some queries that we are unable to translate. Should there be some other table reference or mention of any aggregate functions in the REPLACE target list we can not translate to an equivalent UPDATE. However this does not preclude us from translating all REPLACE queries. Thus the steps involved in the translation are as follows:

1. Firstly the table name given in the REPLACE statement is checked to see whether it is in fact an alias. If it is we replace it with the table name it represents.

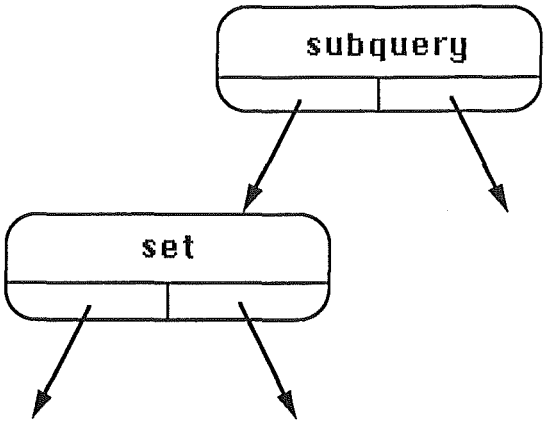
2. The first entry into the parse tree is made.



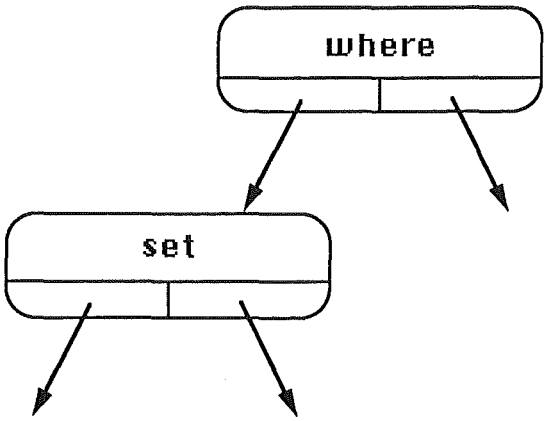
3. Next the SET clause is connected to the parse tree.



4. Finally if there is some restriction placed on the update we place the restriction here also. Note that as the WHERE clause of the UPDATE statement places restriction in terms of a search_condition we must first modify the WHERE clause to an algebraic subquery if there is reference made to some other table. If there is no such reference we are free to append the WHERE clause unaltered.



OR



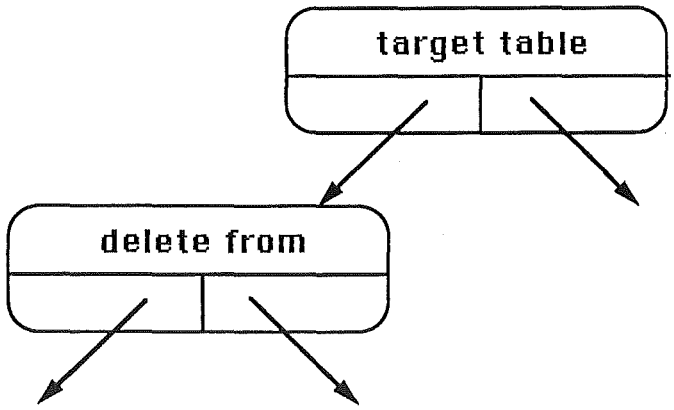
As the UPDATE statement differs quite dramatically from the REPLACE statement we are faced with having to modify the query should the need arise. And what course of action do we take if the REPLACE can not be translated? We must ignore the update request as a whole and assume that the update may only be done as requested as a whole or not at all. In the current implementation this is yet to be implemented.

Section 10

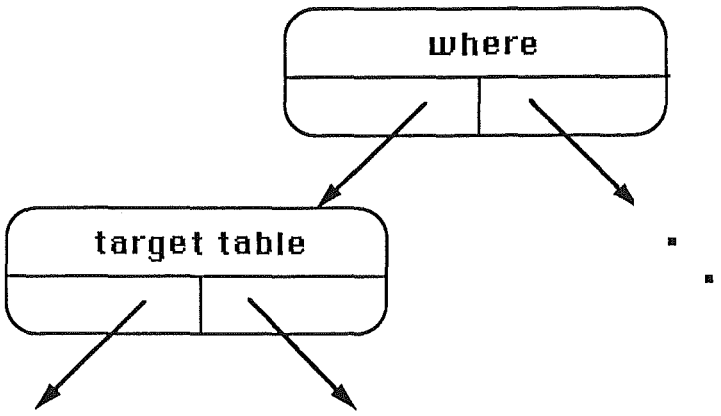
Data Deletion

Here we consider the steps taken to ensure that the data deletion statement is done correctly. Here again, the translation process is not done until the DELETE statement in QUEL has been parsed completely. The steps involved are thus:

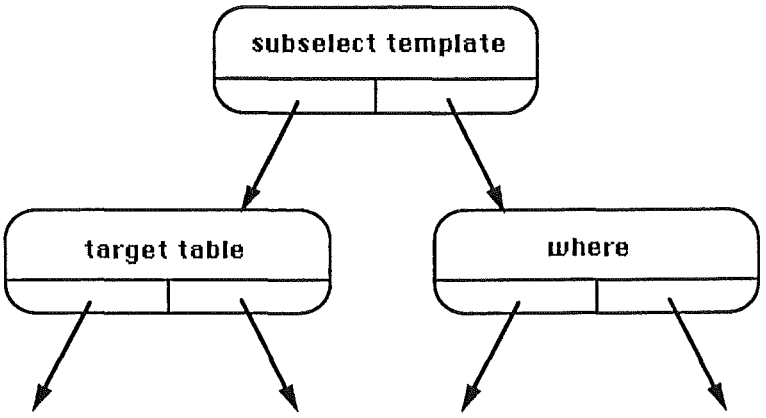
1. Upon reading the name of the target table we construct the trivial case of a table-wide deletion.



2. If there is some WHERE clause that comes with the query then this constraint is checked to see if there is any reference to another table. If there is not then the WHERE clause may be appended without any modifications.

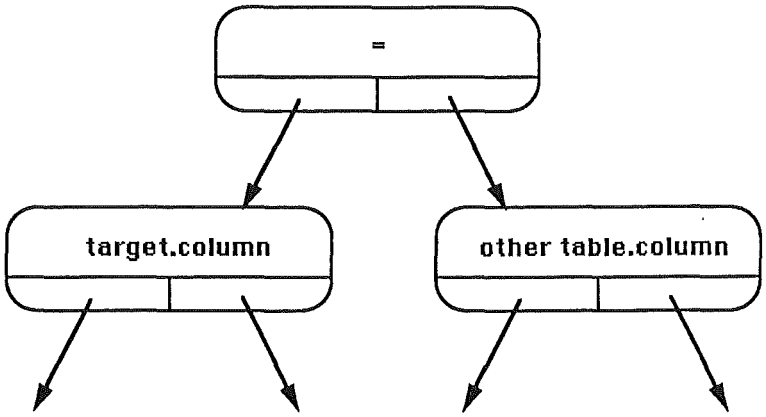


If there is a reference to some other table then the WHERE clause must be modified. The join predicate, once found, is then used in a subquery template (i.e. "where ... comp_op any (select ... from ..." is the template used for a subquery).

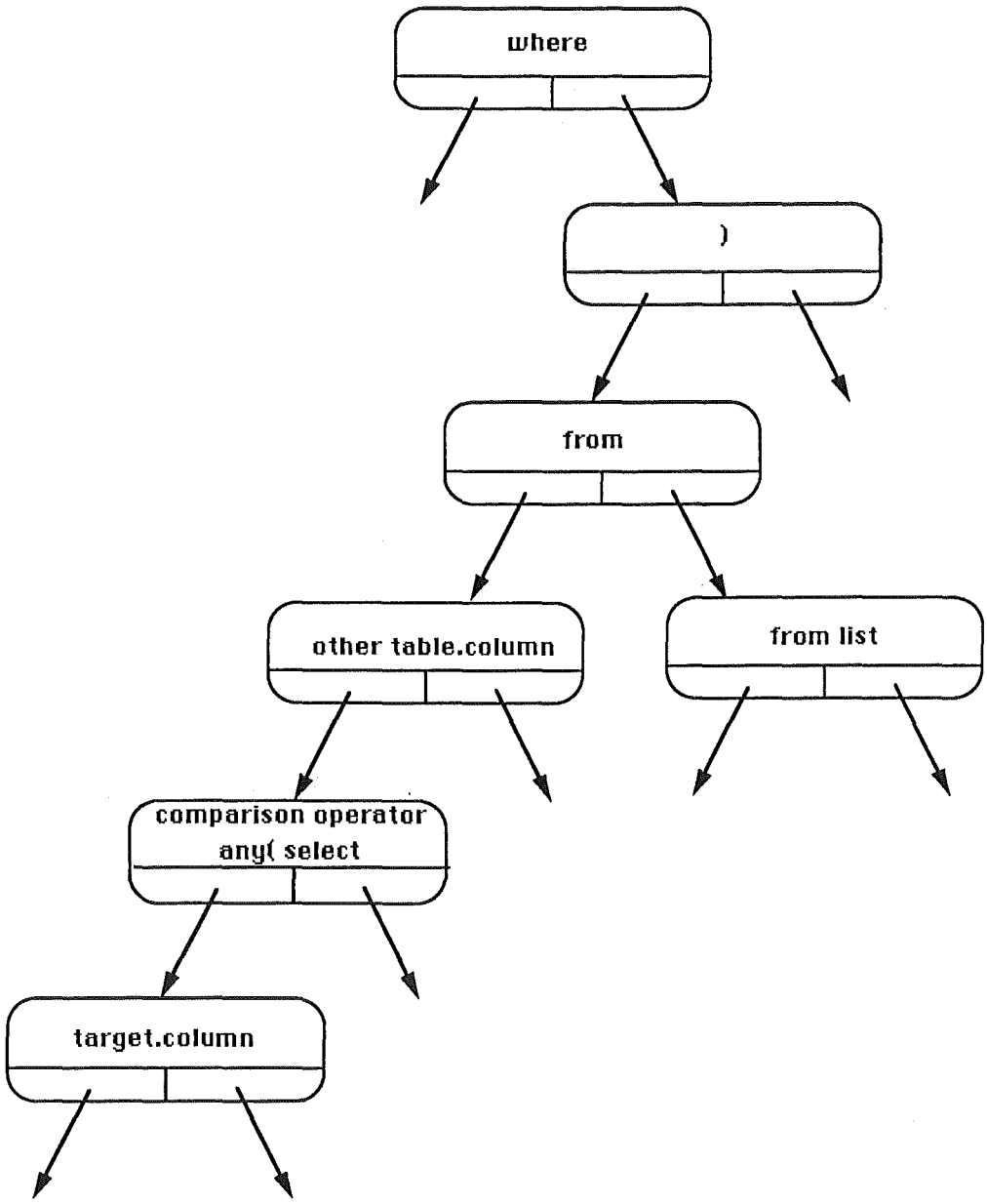


To find the join condition is not a simple matter. It involves intricate navigation through the WHERE clause parse tree looking for a comparison operator. Once the comparison operator is found then the two children are searched further for a table reference. If there are only constants in one of the scalar expressions, then the search is abandoned for that part of the parse tree and a null result returned. If there is a table reference made in the subtree then it is compared with the main relation. If they are the same then the search is abandoned for that part of the parse tree. Otherwise we have found the join predicate and hence return this subtree. At the root will be the comparison operator. The left child will contain the subtree reference of the target table and the right child will contain the table to which it is compared to.

e.g.



Once the join condition has been found then the reference to the table references on either side of the comparison operator is then placed into the subselect template. The template is composed thus:



The Subquery Template

Thus we are able to generate all subqueries in this manner. Note how we generate a template only if we are able to find some linkage condition involving the main query. Should the QUEL predicate contain join predicates between other tables then they are ignored. This has the potential for error in that the QUEL query might specify some predicate where only an unrelated join predicate is given in the WHERE clause.

```
e.g.   delete A
       where B.x = C.x
```

Should such a query arise then we are left defenseless. As there is no reference to the target table we are unable to link it in the predicate at all. To remove this would be erroneous as the condition itself may be necessary. The query is stating that if B.x = C.x then we are to delete the whole table A. Otherwise we are to leave A untouched. Hence there is no easy answer for this case. This is a somewhat meaningless type of query, attempting to achieve some form of selection control but we regret that should such a query be passed to the translator it will be passed through without any modification.

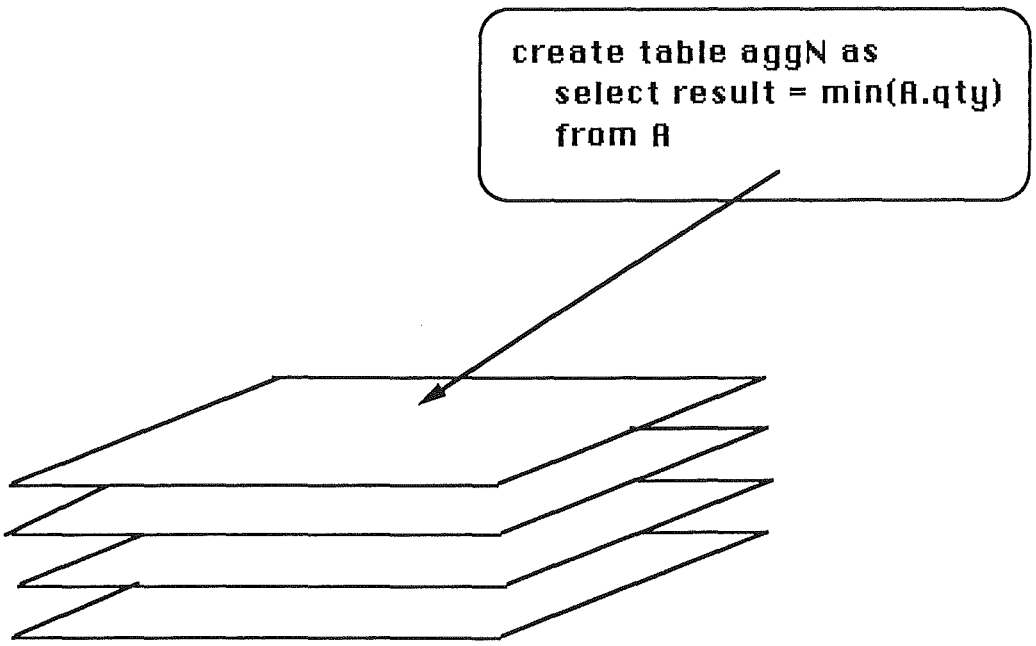
Section 11

Aggregate Functions

Although we were not able to implement the translation of aggregate functions completely at the time this document was produced we nevertheless detail the implementaion strategy that has been started and to be completed.

As was mentioned in the design guide, the only way we may translate an aggregate function to the equivalent set function and retain the power of the former is to translate every occurrence of an aggregate function into a new table. To implement this a stack of parse trees was constructed. Each frame on the stack contains an SQL declaration for creating a new table. It was envisaged that as Yacc recognised an aggregate function in the input stream, we would translate it immediately. Then this new table definition would be pushed onto the AGG-stack and the original function reference would be replaced with a reference to the appropriate frame on the AGG-stack. Then the necessary linkage condition would be appended to the original WHERE clause.

Once the QUEL input was exhausted and the translation to SQL completed, one would then dump the contents of the AGG-stack before outputting the contents of the main query. Then after the main query was dispatched, one would then issue a series of DROP commands to the SQL host to delete the newly created "aggregate tables".



The translation of "... min(A.qty) "

In the above example above we see what happens to a query that involves a simple aggregate function. The aggregate function will be translated to become a new table. The declarations for this newly CREATE'd table is then pushed onto the AGG-stack. The reference to the aggregate function will then be translated to become a reference to "... aggN.result".

The general principle for translating an aggregate function reference in QUEL is thus:

- replace all BY clauses by the equivalent GROUP BY of the resulting SELECT subquery.
- append any WHERE clause to the subquery with no modifications.
- convert any unique aggregate functions to the distinct set functions e.g. "... avgu(..." becomes "... avg(distinct ...", etc
- give a unique name to the resulting column in which the results of the query will be stored.

e.g. "... select result = avg(distinct ..."

- finally put all the columns referenced in the original BY clause into the subselect clause (This is a language restriction placed upon us because in SQL one may only select columns that have a single value per group).

e.g. consider the occurrence of the following aggregate function

```
countu(A.x by A.y, B.x
       where B.x < 42)
```

which will be stored on the aggregate function stack as

```
create table aggN as
     select result=count(distinct A.x),A.y,B.x
     from A, B
     where B.x < 42
     group by A.y, B.x
```

This scheme was developed as a general algorithm for handling the occurrence of any aggregate function anywhere within a scalar function in the QUEL query. The algorithm has been implemented in part (mechanisms exist for creating the new table, pushing and popping the tables on and off the stack) but not completed. With the existing primitives in place it would not be difficult to complete the algorithm.

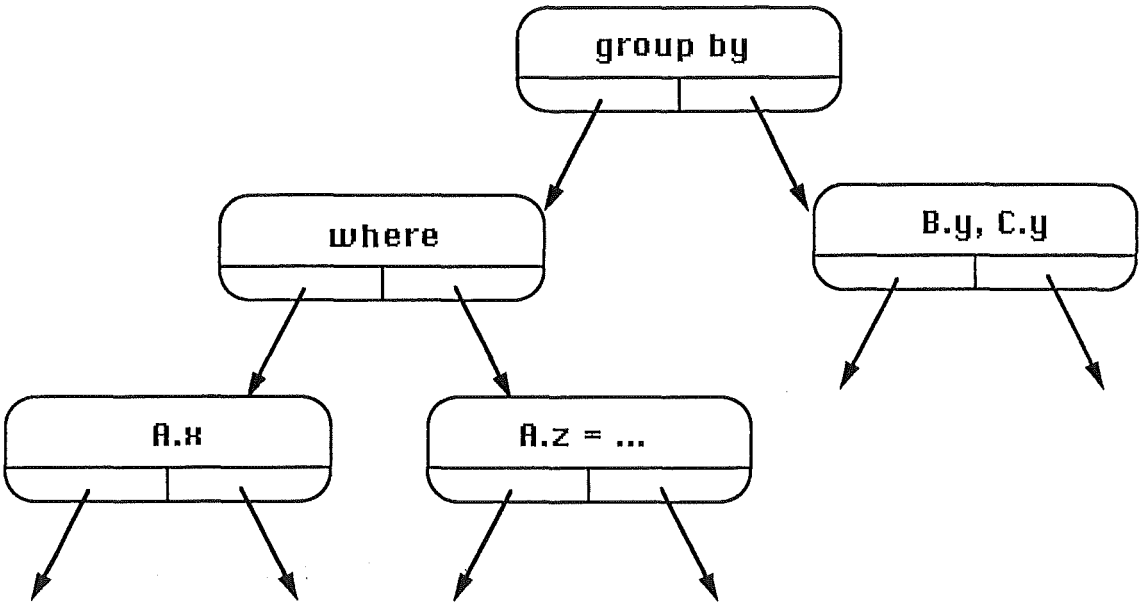
The implementation as it stands is able to store the translated query thus:

1. Firstly the argument to the aggregate operator is modified to a partly translated form where the BY clause is converted to the GROUP BY clause and is then placed after any WHERE clause.

   e.g. when Yacc recognizes the following as an argument to the aggregate function

   ```
   "... A.x by B.y, C.y where A.z = ..."
   ```

the following partial translation is made into a parse tree thus:



2. Yacc then reduces the aggregate operator argument back to the aggregate function statement. We are then able to take the name of the aggregate operator and the information contained in the partially translated aggregate function argument and complete the translation of the aggregate function. To complete the translation we build a CREATE template (much the same way as for the subquery template) detailing the new name of the function, the columns to be selected and the tables they are to be selected from. To build the result column we put take the original argument (the left most descendent in the partially translated function argument) and use it as the new argument to the set function.

3. Once the tree is created containing the declaration for making a new (temporary) table, we then add it to our AGG-stack.

Once the translation is completed and ready to output, we then go through the following steps:

1. Dump the contents of the AGG-stack
2. Dump the main query
3. Issue a DROP command for each temporary table CREATEd.

Thus we have the implementation for translating aggregate functions. At the time this document was written routines for creating, storing and outputting aggregate functions were in place. There were no routines for setting up the appropriate linkage conditions.

Possible Extensions

In this project we have restricted the range of translation to the data manipulation sub-languages to highlight the differences between QUEL and SQL. There is scope for developing this translator further.

Firstly, to make the translator complete we could extend the range to include the data definition language (DDL). The application of the translator could then be to interface directly with a heterogeneous distributed database system. We could interface with such a system as it stands but this would only be of use in systems where resource allocation was strictly policed by the foreign system.

Also we could consider the translation in the reverse direction. This would be mutually beneficial to the user and the heterogeneous distributed database system as then the user that interfaces with said network could actively participate in resource sharing.

Finally the extension that has the most interest for this author would be to interface with a graphical user front-end that outputs QUEL in ASCII character form. Such interfacing would merely involve the setting up of processes on the Sun's UNIX environment that would control the information traffic between this translator, the graphical front-end and our version of RTI INGRES.

Conclusion

As SQL is about to be made the industry standard it was felt that a translator for translating from QUEL to SQL would be needed. Here we have presented a translator that translates the QUEL data manipulation operators into the equivalent SQL data manipulation operators.

We have detailed the design decisions that were made in translating the DML and looked at how said decisions were implemented. We have also presented a proposed algorithm for translating aggregate functions to set functions.

There are possibilities for extending this project further. Among the possibilities we have considered is the relatively simple task of interfacing with a graphical front-end currently being undertaken as a Masters project.

Finally, we have found that in implementing the translator there exists a class of APPEND queries that may not be readily translated to equivalent UPDATE queries. We know that by Codd's theorem of functional equivalence this should not be the case.

We attribute this deficiency to the fact that SQL is not a full implementation of relational algebra. In its attempt to become the relational Jack-of-all-trades it has become something of a master-of-none. We also attribute some of the blame to the fact that SQL is somewhat idiosyncratic.

REFERENCES

[OWRA87] Owrang and Miller;
'Translation of queries to account for direct communication between different DBMSs";

AFIPS Conference Proceedings - vol.56, (1987)

[SURE76] Su and Reynolds;
"Conversion of High Level Sublanguage Queries to Account for Database Changes";

Proceedings of the 2nd International Conference on VLDB pp 143-157 (1976)

[KATZ82] Katz and Wong;
"Decompiling CODASYL DML into Relational Queries";

ACM TODS, vol. 7 no. 1, (1982)

[DATE86] Date, C.J.;
An Introduction to Database Systems;

Addison-Wesley Publishing Company Inc, (1986)

[DATE83] Date, C.J.;
An Introduction to Database Systems;

Addison-Wesley Publishing Company Inc, (1983)

[SCHR85] Schreiner and Friedman,Jr;
Introduction to Compiler Construction with UNIX;

Prentice-Hall Inc, (1985)

[CODD70] Codd, E.F;
"A Relational Model of Data for Large Shared Data Banks";

CACM vol 13 no. 6, (1970)

[CODD72] Codd, E.F;
"Relational Completeness of Data Base Sublanguages";

Data Base Systems,
Courant Computer Science Symposia Series, vol 6 (1972)

[ASTR76] Astrahan et al;
"System R: Relational Approach to Database Management";

ACM TODS vol 1 no. 2 (1976)

[CHAM74] Chamberlin and Boyce;
"SEQUEL: A Structed English Query Language";

ACM SIGMOD Workshop on Data Description, Access, and Control (1974)

[FERR87] Ferrante, L;
"A Comparison of the ISO Working Draft Standard for SQL and a Commercial Implementation of SQL";

SIGSMALL vol. 13 no. 3 (1987)